Algoritmos y Estructura de Datos: Examen 2 (Solución)

Grados Ing. Inf. y Mat. Inf. Junio 2011

Departamento de Lenguajes, Sistemas Informáticos e Ingeniería de Software

Apellidos: .....

Nombre: .....

DNI / NIE: ..... **Núm. matrícula:** .....

Normas

- Este examen consta de **5 preguntas** en **6 páginas**.
- La puntuación total del examen es de **10 puntos**.
- La duración total del examen es de **60 minutos**.
- El examen debe contestarse **en las hojas que se proporcionan**.
- Deben rellenarse los campos obligatorios **apellidos, nombre, y DNI/NIE**.
- Las calificaciones provisionales de este examen se publicarán en el Aula Virtual el **22 de junio de 2012** junto con las soluciones. La fecha de la revisión de este examen es el **28 de junio de 2012**. La hora y lugar de dicha revisión se anunciará en el Aula Virtual.
- En las preguntas con varias opciones, **sólo hay una respuesta válida por pregunta**. En este caso toda pregunta en que se marque más de una respuesta se considerará incorrectamente contestada y toda pregunta incorrectamente contestada restará del examen una cantidad de puntos igual a la puntuación de la pregunta dividido por el número de alternativas ofrecidas en la misma menos uno. Es decir, una respuesta incorrecta en una pregunta de un punto con cuatro alternativas resta $\frac{1}{3}$ de punto.

(1 punto) 1. Se desea dotar a Eclipse de una caja de diálogo en la que cuando el programador escriba el nombre de una clase Eclipse abra una pestaña con el fichero en el proyecto en curso donde se define dicha clase. Se desea que la búsqueda tenga la mejor complejidad posible. **Se pide:** Indicar de entre las siguientes posibilidades la estructura de datos que mejor se adecuaría al problema:

   (a) Una cola con prioridad implementada mediante una lista ordenada.

   (b) Una cola con prioridad implementada mediante un montículo.

   (c) Una función finita implementada mediante una tabla de dispersión. ✔

   (d) Una función finita implementada mediante una lista desordenada.

(1 punto) 2. **Se pide:** Indicar de entre las siguientes posibilidades la complejidad en peor caso del algoritmo de ordenación por inserción «insertion sort».

   (a) $O(1)$

   (b) $O(\log(n))$

   (c) $O(n\log(n))$

   (d) $O(n^2)$ ✔

(2 puntos) 3. El siguiente código Java es un fragmento de una clase `HeapPQ<K,V>` que implementa el interfaz `PriorityQueue<K,V>` mediante un montículo. (El interfaz `PriorityQueue<K,V>` está disponible en el Apéndice A.1.) La clase `HeapPQ<K,V>` tiene tres atributos. El atributo `v` es el vector de entradas. El atributo `size` guarda el tamaño de la cola con prioridad. Finalmente `comp` almacena el comparador de claves. En el fragmento sólo se muestra un constructor y un método `misterio`:

```
public class HeapPQ<K,V> implements PriorityQueue<K,V> {
  Entry<K,V> [] v;
  int size;
  Comparator<K> comp;

  public HeapPQ(int n, Comparator<K> c) {
    v    = (Entry<K,V>) new Object[n];
    size = 0;
    comp = c;
  }

  protected void misterio(int pos) {
    Entry<K,V> tmp;
    while( pos > 1 &&
           comp.compare(v[pos].getKey(),v[parent(pos)].getKey()) < 0 ) {
      tmp            = v[pos];
      v[pos]         = v[parent(pos)];
      v[parent(pos)] = tmp;
      pos            = parent(pos) ;
    }
  }

 /* más métodos aquí */
}
```

El método `misterio` se invoca pasando como parámetro la posición (índice) de la última entrada almacenada en el vector. **Se pide:** Indicar de entre las siguientes posibilidades la que revela correctamente el nombre y función del método `misterio`:

(a) Se trata de «up-heap» que restablece la propiedad de orden entre claves tras una inserción. ✔

(b) Se trata de «down-heap» que restablece la propiedad de orden entre claves tras un borrado.

(c) Se trata de «up-heap» que restablece la propiedad de orden entre claves tras un borrado.

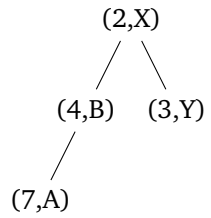
(d) Se trata de «down-heap» que restablece la propiedad de orden entre claves tras una inserción.

(3 puntos) 4. La siguiente figura muestra el contenido del vector de una implementación mediante montículo de una cola con prioridad. El interfaz `PriorityQueue<K,V>` está disponible en el Apéndice A.1.

null	(2,X)	(4,B)	(3,Y)	(7,A)	null	null	null

Se pide:

(a) (2 puntos) Dibujar el árbol (casi)completo representado por el vector.



(b) (1 punto) Dar la secuencia de entradas que se obtendrían al invocar el método `removeMin()` consecutivamente hasta vaciar la cola con prioridad.

(2,X), (3,Y), (4,B), (7,A)

(3 puntos) 5. **Se pide:** Implementar en Java el método

```
public void inOrder(BinaryTree<E> t)
```

que toma como parámetro un árbol binario y muestra por pantalla la secuencia de elementos almacenados en los nodos según un recorrido en «inorden». Los interfaces `BinaryTree<E>` y `Tree<E>` están disponibles respectivamente en el Apéndice A.2 y en el Apéndice A.3.

```
public void inOrderAux(BinaryTree<E> t, Position<E> p) {
    if (t.hasLeft(t,p))  inOrderAux(t, t.left(p));
    System.out.print(p.element().toString() + " ");
    if (t.hasRight(t,p)) inOrderAux(t, t.right(p));
}
public void inOrder(BinaryTree<E> t) {
  if (!t.isEmpty()) inOrderAux(t,t.root());
}
```

A. Código de apoyo

A.1. Interfaz `PriorityQueue<K,V>`

```
package net.datastructures;

public interface PriorityQueue<K,V> {

  /** Returns the number of items in the priority queue. */
  public int size();

  /** Returns whether the priority queue is empty. */
  public boolean isEmpty();

  /** Returns but does not remove an entry with minimum key. */
  public Entry<K,V> min() throws EmptyPriorityQueueException;

  /** Inserts a key-value pair and return the entry created. */
  public Entry<K,V> insert(K key, V value) throws InvalidKeyException;

  /** Removes and returns an entry with minimum key. */
  public Entry<K,V> removeMin() throws EmptyPriorityQueueException;
}
```

A.2. Interfaz `BinaryTree<E>`

```
public interface BinaryTree<E> extends Tree<E> {
  /** Returns the left child of a node. */
  public Position<E> left(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns the right child of a node. */
  public Position<E> right(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns whether a node has a left child. */
  public boolean hasLeft(Position<E> v) throws InvalidPositionException;

  /** Returns whether a node has a right child. */
  public boolean hasRight(Position<E> v) throws InvalidPositionException;
}
```

A.3. Interfaz `Tree<E>`

```
package net.datastructures;
import java.util.Iterator;
/**
 * An interface for a tree where nodes can have an
 * arbitrary number of children.
 * @author Michael Goodrich
 */
public interface Tree<E> {
  /** Returns the number of nodes in the tree. */
  public int size();

  /** Returns whether the tree is empty. */
  public boolean isEmpty();

  /** Returns an iterator of the elements stored in the tree. */
  public Iterator<E> iterator();

  /** Returns an iterable collection of the the nodes. */
  public Iterable<Position<E>> positions();

  /** Replaces the element stored at a given node. */
  public E replace(Position<E> v, E e)
    throws InvalidPositionException;

  /** Returns the root of the tree. */
  public Position<E> root() throws EmptyTreeException;

  /** Returns the parent of a given node. */
  public Position<E> parent(Position<E> v)
    throws InvalidPositionException, BoundaryViolationException;

  /** Returns an iterable collection of the children of a given node. */
  public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is internal. */
  public boolean isInternal(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is external. */
  public boolean isExternal(Position<E> v)
    throws InvalidPositionException;

  /** Returns whether a given node is the root of the tree. */
  public boolean isRoot(Position<E> v)
    throws InvalidPositionException;
}
```